



National Research
Council Canada

Conseil national
de recherches Canada

ERB-1008

Division of
Electrical Engineering

Division de
génie électrique

Debugging Multitask Programs

W.M. Gentleman and D.A. Stewart
September 1987

88-04452/

Copyright 1987 by
National Research Council of Canada

Permission is granted to quote short
excerpts and to reproduce figures and tables
from this report, provided that the source of
such material is fully acknowledged.

Additional copies are available free of
charge from:

Editorial Office, Room 301
Division of Electrical Engineering
National Research Council of Canada
Ottawa, Ontario, Canada
K1A 0R8

Copyright 1987 par
Conseil national de recherches du Canada

Il est permis de citer de courts extraits et de
reproduire des figures ou tableaux du
présent rapport, à condition d'en identifier
clairement la source.

Des exemplaires supplémentaires gratuits
peuvent être obtenus à l'adresse suivante:

Bureau des publications, Pièce 301
Division de génie électrique
Conseil national de recherches du Canada
Ottawa (Ontario) Canada
K1A 0R8

DEBUGGING MULTITASK PROGRAMS

W.M. Gentleman and D.A. Stewart

ERB-1008

NRCC No. 28252

SEPTEMBER 1987

TABLE OF CONTENTS

	Page
ABSTRACT/RÉSUMÉ	v
1. DEBUGGING TECHNIQUES	1
2. WHY MULTI-TASK PROGRAMS ARE DIFFERENT	4
3. A TROUBLESHOOTING GUIDE	5
3.1 Examination of the state of tasks	5
3.2 Examination of standard system data structures	6
3.3 Using a spy task	6
3.4 Check for stack overflow	6
3.5 Check for storage pool corruption	7
3.6 Check for deadlock	7
3.7 Check for unexpected implications of preemptive priority	8
3.8 Consider the consequences of task destruction	8
3.9 Look for unnecessary sequentialization	9
3.10 (Look for critical races and unsuspected shared variables)	9
4. FEATURES FOR DEBUGGING SEQUENTIAL PROGRAMS	9
5. FEATURES FOR REALTIME OR PARALLELISM	12
6. USER INTERFACE ISSUES	13
7. CONCLUSIONS	15
REFERENCES	15



ABSTRACT

Techniques used to debug sequential programs can, of course, also be used for debugging concurrent or even parallel programs. Especially when concurrentness or parallelism is explicit in the program, or when the program must respond to asynchronous external events, however, these techniques may be insufficient. Debugging such programs is difficult because timing of events can be critical. This is particularly true for performance bugs, even more than for correctness bugs. Whereas the traditional debugger for sequential bugs is based on the "stop the world, I want to get off" breakpoint, in a multi-task program normally one wants to stop a specific task, and indeed halting all the processors may be impossible. In addition to breakpoints, other techniques have been found useful.

This paper describes the techniques supported by the new debugger currently being built for use with the Harmony¹ operating system. Three categories of features are discussed: non-realtime (traditional) tools, realtime (multitasking multiprocessor) tools, and the user interface.

RÉSUMÉ

On sait fort bien que les méthodes utilisées pour la mise au point des logiciels séquentiels peuvent aussi servir à la mise au point des logiciels simultanés ou même parallèles. Toutefois, ces méthodes ne suffisent pas dans certains cas, en particulier lorsque la simultanéité ou le parallélisme est explicite dans le logiciel ou lorsque celui-ci doit répondre à des événements externes asynchrones. Le mise au point de ce type de logiciel est difficile car la synchronisation des événements peut être un facteur critique. Cela est surtout vrai dans le cas des défauts de performance, plus que dans le cas des défauts d'exactitude. Les programmes traditionnels de correction des défauts séquentiels sont fondés sur la coupure totale au point d'interruption; dans un logiciel multitâche, on préfère normalement n'arrêter qu'une seule tâche et il peut même être impossible d'arrêter tous les processeurs. On a développé d'autres méthodes commodes, qui s'ajoutent à celle des points d'interruption.

Le présent article décrit les méthodes compatibles avec le programme de mise au point en cours de développement pour le système d'exploitation Harmony¹. Trois types de fonctions sont décrites: les outils en temps non réel (les outils traditionnels), les outils en temps réel (multiprocesseur multitâche) et l'interface utilisateur.

¹Mark reserved for the exclusive use of Her Majesty the Queen in right of Canada by Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée. / Marque déposée réservée à l'usage exclusif de sa Majesté du chef du Canada par Canadian Patents and Development Ltd./Société canadienne des brevets et d'exploitation Ltée.

DEBUGGING MULTITASK PROGRAMS²

W.M. Gentleman and D.A. Stewart

1. DEBUGGING TECHNIQUES

Most programs require debugging. This is particularly true for complex programs, which realtime programs or parallel programs tend to be. Although this statement is trite, few programmers have a disciplined methodology of debugging and few schools teach how to debug programs. Before focussing on the topic of this paper, let us look more generally at how debugging is done today.

All too often the debugging technique practiced can best be described as wild optimism. A large program is written in its entirety, and then debugging is attempted by running it over and over with trial data, looking for aberrant behaviour. Initially the program often fails spectacularly, but as bugs are found and fixed, a stage is eventually reached where the program usually behaves as expected. Sande (1) once called this "debugging by friction" — if the program is run through the machine often enough, maybe the bugs will wear off!

Criticism of this approach usually focusses on the fact that one can never be sure of finding the last bug, that one can never be sure the program is correct. Instead, we prefer to make the observation that this approach is highly inefficient of the programmer's effort. The essential steps of debugging are, first, detecting that something has failed and, second, tracing back through the derivation of that something in order to locate the cause of the failure. Given all the output of a complete program, it is often exceedingly difficult to detect that it has failed, unless the failure is something spectacular such as an address exception. Faced with a complex mass that has failed, the programmer often has an immense job tracking back from the symptom to the cause. Trying to debug everything at once often means that several bugs interact, some disguising the effects of others and making cause and effect relationships difficult to establish. Moreover, even after identifying bugs found at this stage, properly correcting them often results in major redesigns that throw away much of the code that was written, clearly showing that effort was wasted. We believe that this approach stems from a misguided belief that the program must be written in its entirety for the programmer to grasp fully its interlocking intricacies and to settle design issues, combined with the pious hope that maybe this time the coding will all be correct. Simplistic "program proof" can actually exacerbate this effect. We believe instead in planning for debugging.

The key idea in planning for debugging is that of using systematic testing. Typically, for a given program, we want to develop a strategy whereby a sequence of tests can establish, at each step, that part of the program is probably working correctly, which can be used at the next step to show that another part of the program is probably working correctly. Unit test, i.e., separately testing individual components, is one such way. Another simple and popular way to do this is by recognizing that most programs have milestones, which are points in the execution where some phase of the computation is complete and testable properties should hold for the data structures that have been built and for the values in these data structures. Even just reaching a milestone can be significant, but the testable properties are more important. Verification and display of these properties are probably not part of the normal output of the program running in production mode, but debugging code can be used to do this when the program is run on test data for which the expected properties can be derived. Note that working with such test data alleviates the first step of debugging, that of recognizing when something is wrong. Note also that proceeding from milestone to milestone limits how far back the programmer must search to locate the source of the problem. Note that often milestones can be chosen at stable points in the program so any parallelism or concurrentness, even indeterminacy, used in advancing from the previous milestone can be ignored in subsequent analysis. Note finally that often the probable correctness of milestones can be established during development while later code still only exists as stubs.

²A version of this paper appears in the *Proceedings of the Army Research Workshop on Parallel Processing and Medium Scale Multiprocessors*, Stanford, CA, January 6-8, 1986.

We have stressed "probable correctness" in the previous paragraph because it is well known that the dominant proportion of the persistent bugs in large programs are due to the programmer incompletely recognizing the complexity of the situations his code must handle, indeed that often this failing is also present in the problem specification (2). Thus systematic testing, like proof of correctness, will almost certainly have the deficiency of falling only within the domain of the problem as the programmer understood it (3). However, once the machinery for testing is in place, and confidence established in the normal case properties of the program, we are in a much better position to examine those properties in realistic situations or in situations where anomalous behaviour has been observed. Simply examining the program state at milestones often reveals situations that the program will not be able to handle in its subsequent computation.

Typically, the examination of properties described above is done manually. This can be valuable for exposing unanticipated aspects of the data, but is tedious and error prone. Consequently, a better approach is to automate the examination, by the programmer making assertions in the code as to what properties should hold at indicated milestones, and providing code in the form of audit routines that can be executed on intermediate data structures to verify the assertions.

To make this discussion more concrete, let us consider some specific examples. A program that constructs a doubly linked list of records and then uses the list has a possible milestone at the point where the construction of the list is complete. Properties that might be tested include those of individual records, such as that values in various fields are plausible and that predecessor pointers in successor records point back to the record itself. Other properties that might be tested include those of the list as a whole, such as that the list terminates properly, that all necessary records are in fact on the list, and that if the list is supposed to be ordered with respect to some field then that it is so ordered. Notice that properties such as the preconditions, postconditions, and loop invariants that are used in proof of program correctness are also candidates to examine, but they are not the only properties one would look at.

Another example might be an engineering program, based on the finite element method, where a stress matrix is set up, then solved for various right-hand sides, perhaps with perturbations to the elements of the matrix. A possible milestone could be where the matrix has been formed, and a property that might be examined is that the eigenvalues of the matrix are sufficiently far from zero. By construction the matrix will be symmetric and positive semidefinite, so what this property is really testing is whether the chosen basis actually spans the space of solutions satisfactorily. What is interesting about this example is that the property is not immediately evident in terms of data normally available at the milestone (indeed the audit routine must do more work than the next stage of the normal computation). Even after the audit calculation, judgement is required to decide if all is well.

Returning to general techniques of debugging, one that is often suggested is simulation. In many cases what is meant is that a separate program should be written that is a simplification of the real program. This is sometimes suggested because the real program is too difficult to understand, or too expensive to conduct experiments on, but more commonly it is suggested because it is hoped that the simulation can be programmed more quickly and more cheaply than the real program, so that design decisions for the real program can be anticipated and evaluated in the simulation. Though such an approach is frequently suggested, it is hard to find cases where it has been followed, and even more difficult to find cases where it has been effective. There are a number of reasons for this. Clearly such a simulation will not find detail coding errors but at best might identify design shortcomings. However, it is exceedingly difficult to ensure that the simulation faithfully reflects how those design decisions will affect the real program, especially before the real program exists. A faithful simulation is likely not to be cheap, in particular requiring a scarce resource — the people capable of building the real program. Few organizations can tolerate the delay of building a comprehensive simulation before building the real program. Finally, because the simulation is not the real program, it usually cannot do the full job of the real program, but can only be used in a subset of the situations the real program might face, which means it is likely to share design oversights with the real program and not to be exercised so as to reveal them. The two situations where simulations have proved effective are when the simulation is intended to measure performance of a real program by adding instrumentation that the real program does not contain, for

instance, because the hardware does not provide the information sought, and when hardware must be simulated because it is not available.

(The current round of promotion of prototyping tools for microcomputer applications may obsolete the comments in the above paragraph, at least for user interface design, because the tools make the cost of simulating the interface orders of magnitude cheaper than the cost of writing the actual program. Whether these tools are as effective in practice as claimed remains to be proved.)

In the context of parallel computing, however, what is usually meant by simulating the parallel program is something quite different. Programmers new to the world of concurrentness are often uncomfortable with the loss of control over the order in which things get done and want to debug the data structures and algorithms of their programs in a sequential mode, before addressing problems, such as deadlocks or critical races, which might arise from their use of multiple processors. In many parallel execution environments, whether with the anonymous tasks often used with the shared memory model or with the explicit tasks used with message passing systems, this is actually quite easy to do as the tasks are distributed over the available processors, and if only one processor is made available (and it has enough resources to run the program at all), then the same code, unchanged, is a sequential program. In our experience, this is not a useful thing to do.

Part of the reason for such a heretical statement is the observation that even on a single processor, the context switching among concurrent tasks induces much the same unpredictability of order of execution that caused the discomfort in the first place, and escaping from this can require such major code changes that one is not really debugging the same program. A deeper issue is that we observe that with good debugging techniques, working in appropriate programming abstractions (such as the explicit multitasking system based on remote procedure call that we use), the classic hazards of concurrent programming are rare. Indeed, the artificial sequentialization of things that were intended to run in parallel induces effects much harder to anticipate and understand. It really is much nicer, for instance, to have a debugger running in a separate processor, unaffected by the whims of the program being debugged.

What does happen when a parallel program is forced to run on one processor is that an amazing number of things that are seriously wrong work accidentally because the forced sequentialization resolves critical races, hides mistakes in preemptive priorities, etc. Consequently, the program gets far beyond where the real problem lies before exhibiting suspicious behaviour. Even just two processors is often enough to alert the programmer to the problems, whereupon such problems are not hard to find or to cure.

Of all debugging tools, the most important, in our experience, is a good interactive debugger used well. Because debuggers are unfashionable in certain parts of the computer science community, this statement needs amplification. The argument against debuggers is usually that they are unnecessary and promote sloppy work because programs should be designed and coded to be correct. This argument is based on the naive view that a sufficient alternative is to prove programs correct. Program proof is, of course, a useful tool for a programmer to have. However, program proof usually means just that the program implements its specifications. In real programs, the question of what to prove is not trivial. There are few interesting programs with a rigid specification — more commonly the program has an intended objective that could be met in many quite different ways. The engineering technique of breadboarding is essentially aimed at defining the specification from experience with prototypes. Also it is well known that oversights are the most common error both in designs and in mathematical proofs, so a manually created proof cannot be trusted.

A different criticism is that misuse of debuggers wastes time. One advantage of debuggers is that through techniques such as setting switches, correcting values, or patching code, they can shorten the edit-compile-link-download-execute cycle, which can be painful on realtime systems (4). However, it is true that if the debugger is not used in a disciplined way, the programmer can lose track of what has been done and may not update the source code appropriately. Also, unless the programmer uses a systematic strategy to locate and eliminate bugs, there can be a tendency to poke aimlessly at the program, learning little.

A significant aspect of the use of debuggers is to analyze "correct" programs, i.e., those that compute what they were supposed to, but where we are worried about run time or resource usage. This is associated with the idea of a performance bug. Often the resource usage of a program is too difficult to analyze, so using an interactive debugger for "dynamic behaviour monitoring" gives a probe whereby data structures and algorithms can be examined as they work with real data to see whether any bottlenecks can be identified and whether any improvements are suggested.

A legitimate criticism is that too many debuggers are hard to use, indeed hard to remember, thus leading to programmer inefficiency. We will return to this later when we address user interface issues.

2. WHY MULTITASK PROGRAMS ARE DIFFERENT

Debugging multitask programs, whether these are programs exhibiting true multiprocessor parallelism or merely concurrentness, has the reputation of being more difficult than debugging conventional sequential programs. While this has not been our experience, it is true that a number of ways in which multitask programs differ from sequential programs do affect debugging.

A major issue in the design of multitask systems is whether they should be constrained to be deterministic; that is, given the same data, should the program always produce the identical results. A sequential program is always deterministic, but in a multitask program the independent tasks can proceed at different rates at different times due, for instance, to resource availability. We note that in many cases determinacy is not a consequence of the original problem: for example, eigenvectors are only determined up to a scaling factor even when they correspond to an isolated eigenvalue and merely lie in the appropriate subspace for a multiple eigenvalue; a graph may have many minimal spanning trees rooted at the same node; an iterative floating point computation may guarantee to reduce the error below some bound but many different approximations would meet that criterion. In such cases there are many equally good results of the computation: the motivation for requiring determinacy is solely to make debugging easier. (Abandoning determinacy in such cases means a check that the answer is correct must be more sophisticated than merely bitwise comparison with an independently computed result, but rounding error often has similar effects.) Behind the desire for determinacy is the intent to run the program again on the same data, but in programs that interact with the external world this is often infeasible because the external conditions are not exactly reproducible, so other debugging techniques must be used. We should also note that requiring determinacy may rule out more effective algorithms (e.g., chaotic relaxation has sometimes been shown to be faster than practical sequential equivalents) and may even make certain problems impossible to formulate (e.g., realtime interception problems where the choice of whether to extrapolate a trajectory from old data or new depends on when the new data become available).

One approach that does not go as far as determinacy is to insist that all failure handling is under the programmer's control. This often makes it possible to guarantee that, at least as seen at synchronization points, the computation is repeatable except where driven by external events. In particular, it means that unpredictable results cannot occur outside of the programmer's control, for instance by resource exhaustion beneath the programming abstraction. A corollary is that the programming abstraction supports some form of flow control on tasks that produce data or produce other tasks.

The classic bugs ascribed to multitask programs are timing errors: critical races, deadlocks, livelocks, etc. Do these problems really occur and are they serious bugs? The answer to this question depends on the programming abstraction that is being used. Sometimes with a particular abstraction the error cannot happen or sometimes with that abstraction the error is very infrequent. If the abstraction does not support shared variables, for instance (as with a pure message passing abstraction), then there can be no critical race on the update of a shared variable, there can be no unintentional sharing of a variable, and there can be no variables intended to be shared but for which by mistake each task got its own copy. Semaphores are prone to errors of forgetting to signal that the semaphore is free when this is not done textually close to the wait on the semaphore, as well as being prone to loss of resources when a task holding those resources is destroyed, neither of which are common errors when synchronization is instead achieved by remote procedure call blocking communication. (Of course,

errors made by users of an abstraction can be very different from those made by implementers of that abstraction.) In any case, experience with many abstractions suggests that some timing errors, deadlock, for instance, are not serious errors, in that if they occur they are obvious and are readily fixed. On the other hand, critical races for shared variables, especially inadvertently shared variables, are serious because they are hard to detect. Protocol errors are serious because programmers appear to find it hard to anticipate all the implications of repeated passes through a state table or all the situations that can occur. The more subtle experience is that many performance bugs are associated with timing errors. For example, to avoid critical races the execution order of some set of tasks is overconstrained, with consequent loss of potential parallelism.

An important consequence of multitask programs no longer having a single thread of control is that, in understanding the program, there is a shift in emphasis from following the flow of control to concentrating instead on the protocol of interaction among the tasks. In many cases this protocol is over several interactions, not just communication at a single instant in time. Traditional debugger breakpoints are not very helpful here, as the breakpoint event is associated with control flow only and provides only a snapshot of the program at that point. Even simple sequential triggering is not provided. In general, for explicit tasking abstractions based on message passing there is a shift from being code oriented to being object oriented, i.e., task oriented. This means that a breakpoint is usually not usefully tripped when just any task executes that instruction, but only when a specific task does. (There are, of course, exceptions.)

Whereas a sequential program running into a breakpoint stops everything, and the first thought of what might be desirable in parallel computation (regardless of how hard it might be to implement) is a similar "Stop the world, I want to get off!" approach, it turns out frequently to be quite sufficient to stop only the task that tripped the breakpoint (and, of course, those that become blocked because of that task's unavailability). This is most important in realtime situations where critical activities cannot be stopped, but it is also desirable in general, for it means the debugger can just be a group of ordinary tasks within the abstraction and audit routines can also be run as ordinary tasks.

3. A TROUBLESHOOTING GUIDE

For cars, household appliances, and other common products the problems arising in practice are so repetitious that manufacturers supply a troubleshooting guide of standard faults, their causes, and their cures to be tried before looking for something unique. It has been our experience that this is worthwhile for bugs in programs also; for any programming environment, independent of the programmer or the program, the most common faults tend to be similar and the same techniques work to locate and cure them. Here, then, are the troubleshooting techniques we use to find bugs in programs written for Harmony (5), our multitasking, multiprocessor message passing portable realtime operating system.

3.1 Examination of the state of tasks

The first thing to do when a program is halted (at a breakpoint or post mortem) is to examine the state of the relevant tasks in the system, maybe even of all tasks. Although the tasks in our system are explicit, tasks are created dynamically and are not necessarily uniquely associated with textual names, a task being identified instead by a system-supplied numeric `task_id`. Consequently the debugger provides a command to list currently existing tasks and summarize their states. Because there might be a very large number of tasks, and because processors are multiprogrammed and tasks do not migrate among processors, this debugger command takes as an argument a processor number and lists all the tasks currently existing on the specified processor. Another debugger command, given a specific `task_id`, displays the complete state that the system has recorded for that task, including the template from which it was created (what in Ada would be called the task type). Of particular interest is whether the task is eligible to run or is blocked and, if it is blocked, what condition it is waiting for and what queue it is on. In the remote procedure call style of programming, at any time most tasks are blocked awaiting communication, so understanding the blocking conditions and identifying which tasks are attempting to communicate with which others goes a long way toward understanding what the program is doing at that

moment. Because the client-server idiom (6) is used heavily in Harmony programs, it is sometimes of interest to examine the request messages of clients awaiting service or of workers awaiting new work assignments.

3.2 Examination of standard system data structures

The next thing to do once the states of relevant tasks have been examined is to examine standard system data structures. Depending on what we are looking for, these could include the ready queues on a specified processor, the table of tasks on a specified processor that are awaiting interrupts, the storage pool on a specified processor, the directory task's list of symbolically known server tasks, the storage blocks allocated to a specified task, the tasks to which a specified task has opened connections, etc. It is important that the debugger be aware of such system data structures, to be able to display them effectively and to perform simple consistency checks on them. It is also an important part of system documentation to alert the programmer to what system data structures exist that might aid in understanding what his program is doing.

3.3 Using a spy task

There is another general technique of examination that should be mentioned: the use of a spy task. Any attempt to monitor the activity of a program introduces the Heisenberg effect: an artifact of the monitoring is to change the behaviour being monitored. Clearly it is desirable to minimize such interference. The spy task is a powerful technique for doing this. All the monitoring is put into a task different from the tasks being monitored, and this task then relies on sharing address space with the tasks being monitored to enable it to inobtrusively sample their data while they are running. This works particularly well when processors share address space, so the spy task can run on a different processor from the tasks being monitored and only steals a few memory cycles. Running the spy on a different processor also avoids having sampling correlations with the scheduling of the activity being monitored. Even dynamic data structures can be monitored with only the slightest collusion on the part of the monitored tasks; if the monitored task maintains a generation sequence number for the data structure, and the spy checks it before and after reading the structure, then as long as it is the same the spy gets a consistent snapshot. Care must obviously be taken in following pointers lest they change underfoot.

3.4 Check for stack overflow

When we come to actually identifying causes of trouble, there is no question that the most frequent culprit is stack overflow. It is also the most insidious, as the symptoms appear inexplicable: storage pool corruption, variables suddenly changing value without having been assigned to, wild transfers because return addresses have been damaged, in general the programmer's abstraction of the computing environment having been violated. The fundamental issue is that to get many of the benefits of lightweight tasks, such as simple semantics for the behaviour of a single task, requires that many tasks execute in the same address space, in particular that their stacks lie in the same address space. This is, however, not properly supported in hardware, nor by commercial compilers. It would be possible for individual stack frames to be separately obtained from the storage manager, but commercial compilers almost never support this, and indeed it implies undesirable and unpredictable overhead. Consequently, stacks are allocated as fixed-size contiguous storage obtained from the storage pool, and stack overflow detection needs to be by a hardware or software check against the limit of the allocated block. Conventional stack-based systems, such as the Unix implementations of Pascal or C, have a single stack that relies on memory mapping hardware. The stack grows toward address space beyond the bounds of mapped memory, so the stack can be grown effectively without limit so long as memory mapping violations by stack manipulations result in the bounds being extended to map more real memory. Thus, compilers do not check for stack overflow and the only hardware check is the memory mapping violation. Conventional Pascal or C systems without memory mapping hardware grow the stack toward the storage pool, so the stack could grow until it collided with the high water mark of blocks allocated from the pool, and although the stack size is not fixed, similar stack overflow detection is needed to what is needed for the multiple stack case, and many systems similarly do not check.

Although automatic monitoring for stack overflow is not usually provided, and consequently a stack can grow so as to overwrite whatever was adjacent in memory and then shrink back to hide the evidence, it is easy to detect at a breakpoint whether a stack is currently overflowed, and indeed it is easy to provide a runtime function that determines whether the task calling it has currently overflowed its stack. More importantly, use of a program development tool called a stack bouncer guarantees that stack overflow cannot happen. A stack bouncer determines, from compiler provided information or from pseudo-execution, the maximum stack requirement for every function potentially called by the task, then from the call graph determines a bound on the maximum stack requirement of any possible nesting list by summing the individual requirements. Recursion and indirect function calls are resolved through programmer interaction.

Thus checking for stack overflow any time a program is examined and upon any suspicious behaviour using the stack bouncer to confirm that stacks are sufficiently large is a cheap investment that completely eliminates this problem.

3.5 Check for storage pool corruption

Another potential problem worth checking for is corruption of the storage pool. This can be a consequence of stack overflow, using dangling pointers or uninitialized pointers, running off the ends of arrays, allocating record variants that are too small, etc. Each block in the storage pool is preceded by a header indicating block size, whether the block is allocated or free, and, if the block is allocated, a link to other blocks allocated to the same task. If the block is free, the part that would be available to the user if it were allocated contains an unlikely bit pattern. Damage to these fields is likely if there is any corruption of the storage pool, and the debugger can readily check them for consistency (with care, even while the system is running). Tasks owning nearby blocks are prime suspects if corruption is found.

3.6 Check for deadlock

The next potential trouble worth checking is deadlock, not because it is frequent (as discussed above, we find it rare) but because programmers are concerned about it. Deadlock, by definition, is when there is a cycle of tasks waiting on each other. For system level deadlocks in Harmony, this means that each task in the cycle is either send-blocked on another (i.e., it has sent a message that has not yet been received), reply-blocked on another (i.e., it has sent a message that has been received but not yet replied to), or receive-blocked on another (i.e., it is blocked awaiting a message from that other task and only that other task's message will be accepted). Examination of state (especially blocking conditions) and of the system queue data structures readily detects this condition, if it exists, and identifies the tasks involved. Since receive specific is rarely used, such a cycle would likely only involve send-blocked and reply-blocked tasks. The methodology of task structuring in hierarchies with the client-server paradigm, combined with the graphic representation of blocking diagrams for tasks (6, 7), means that cycles do not arise naturally, and are in any case eliminated at design time, except when the programmer is unaware that an apparently innocuous procedure invocation hides intertask communications with consequent possible blocking.

System level deadlocks are, however, not the only type of deadlock that can occur in Harmony. A server can induce a conventional type of deadlock by choosing not to reply immediately to a client's request. For example, a file system server that permits only a single writer on a file can produce a conventional deadlock when task A that has file X open for write attempts subsequently to open file Y for write but task B already has file Y open for write and is also attempting to open file X for write. The deadlock here occurs if the server does not recognize the situation and handle it in the conventional way. Because the deadlock is inside the server, both tasks A and B will be reply-blocked on the server, but the server itself is not blocked, and without examination of the internal state of the server, there is no way to recognize that tasks A and B form a deadlocked cycle. The situation can be more complicated in that the deadlock can arise from conflicting requests for services provided by different servers, so the deadlocked tasks may be reply-blocked on different servers. Moreover, the open architecture of Harmony, where there is no central management of resources, means that conventional deadlock

prevention schemes, such as forcing resources to be acquired in fixed order, are inapplicable. Fortunately, conventional deadlocks like this also seem rare in practice.

A more subtle problem is livelock, where tasks are not actually permanently blocked, but make no progress toward their objective because some task cannot make progress until some other task has performed an action that it does not get around to doing. Because task interaction in Harmony is through message passing, livelock in Harmony usually takes the form of an administrator task, which receives messages from anyone and implements an automaton, getting stuck in an internal state that only a certain message can change despite what other messages the task receives and handles. Examination of system state and system data structures do not help here, except to rule out deadlock as the problem. However, examination of the internal state of the administrator task exposes the problem, and that internal state can usually be probed and changed by sending it messages from the debugger.

3.7 Check for unexpected implications of preemptive priority

One difference between parallelism used to increase computational capacity and concurrentness used for realtime computing is that the scheduling discipline for runnable tasks is rarely an issue for the former, whereas it is often the essence of the latter. Although other scheduling disciplines, such as deadline scheduling, may better meet problem specifications, preemptive priority scheduling is almost universally provided because it is the discipline we know how to implement efficiently and other disciplines can be approximately implemented with it. Preemptive priority scheduling is important for several reasons. From the point of view of high priority tasks, it permits lower priority tasks to be ignored in meeting scheduling requirements. From the point of view of low priority tasks, it permits actions of higher priority tasks to be regarded as atomic. When integrated with the interrupt hardware priority system, it arranges for an interrupt to be held off until the appropriate task is waiting for the interrupt. It can be used to ensure a set of tasks runs in a specific order.

Unfortunately, preemptive priority scheduling does what the programmer specified, not what the programmer expected. Not infrequently tasks at different priorities will be observed to become ready and to run in an unexpected order. When analyzed, what is happening is correct for what was specified, but it is not what the programmer intuitively expected. Unwise choice of priorities can also lead to tasks hogging the processor. Consequently, sequencing problems and performance problems, as well as problems of CPU starvation for tasks, call for careful examination of task priorities.

The integration of interrupt hardware priorities with preemptive priority scheduling for tasks has associated with it a very common configuration error. The interrupt hardware priority level of a particular device is usually set in hardware, e.g., by jumpers, DIP switches, PALs, or PC board traces. The task awaiting the interrupt from that device must run at the corresponding preemptive priority scheduling level, which is declared in software. This is one of many aspects of device drivers where hardware and software must correspond. In general there is no automatic way to confirm that hardware and software configurations correspond, and again and again we have seen it wrong, with baffling effects.

3.8 Consider the consequences of task destruction

Dynamic task destruction has many benefits, but it introduces complications. There are two ways to implement task destruction: poison pill destruction where a task is sent a message to destroy itself and system imposed destruction where the system immediately stops the task from further execution. Poison pill destruction has the disadvantages that the task may ignore the message for some time, that it may improperly clean up thus losing resources, and that the code for destruction is repeated (with minor differences) in every task. Harmony therefore uses system imposed destruction, but that has the difficulty of arranging to clean up properly without knowing exactly what the task being destroyed was doing. By recording resources allocated to a task, it is straightforward for the system to scavenge resources; by recording the connections the task has open, it is straightforward for the system to notify the servers to close those connections; and by recording descendants of a task, it is straightforward for the system to eliminate orphan tasks. What the system cannot do, and thus is

the programmer's responsibility, is to arrange that message protocols will recover appropriately when one of the participants is destroyed. The atomicity of the message passing primitives helps, the connection mechanism is safe, and in particular the client-server paradigm is safe provided that the server checks the result of the reply primitive to ensure the client was not destroyed before being replied to. Programmer-designed custom protocols, however, often are ill-behaved when tasks are destroyed unexpectedly.

A different aspect of task destruction that can cause problems is the fact that task destruction takes finite time, during which other tasks are running, and that it involves message passing between tasks, which is then subject to scheduling priorities. Task destruction in Harmony is complete, in that the destroy primitive does not return until all resources associated with the victim and its descendants have been released, but it is not atomic, and is not even immediate for the victim until the local task manager on the victim's processor has had a chance to receive and act on the destruction request. Unfortunate combinations of priorities can have predictable but awkward consequences, and unexpected intermediate situations can be exposed if the task desiring the destruction uses another task as an agent to call the destroy primitive in order to avoid blocking.

3.9 Look for unnecessary sequentialization

Harmony and related systems have been used extensively by programmers from novice to professional. The realtime programming course at the University of Waterloo alone has been taken by thousands of students over the last decade. What stands out as the most common design error in programs is that the message passing primitives are used in such a way as to cause unnecessary sequentialization. In other words, what is computed is correct, but opportunities for concurrentness or parallelism are prevented, with consequent loss of throughput or responsiveness. This error is called "subroutining", because the remote procedure call communication between tasks is used to induce a single thread of control, exactly as with subroutines. Clues leading one to suspect this problem are indirect, such as unsatisfactory throughput or responsiveness, or casual observation that the ready queues have fewer tasks on them than might be expected. Correspondents and blocking conditions of tasks that are not ready might also suggest the problem, but we must admit that our prime technique for identifying it has been reading the source code.

3.10 (Look for critical races and unsuspected shared variables)

This is included as a final topic because it is a concern in some systems and in some design methodologies. Tasks on the same processor do share address space in Harmony and in many realizations of Harmony a pointer passed from one task to another can be used to access shared data even if the tasks are on different processors, so these problems can potentially happen. The converse problem, where each task accidentally ends up with a private copy of data that was intended to be shared occurs even more easily. Nevertheless, the effect of the message passing design methodology is such that only the most hardened miscreants schooled in the shared memory style of programming ever actually directly share data among tasks unless those data are constant or unless the protocol among the tasks involves a clear transfer of ownership of the data, with only the current owner ever accessing them. Consequently we have titled this topic in parenthesis, because we do not actually recommend taking the trouble to try it.

4. FEATURES FOR DEBUGGING SEQUENTIAL PROGRAMS

In considering design issues for an interactive debugger to be used on multitask programs, the first observation is that much of a multitask program is sequential, and hence the first issues to consider are features desirable for debugging sequential programs.

The key feature in interactive debuggers has been the breakpoint. A breakpoint is a mechanism for stopping the execution of the program when a specified event has occurred. Ideally, there are many kinds of events one might want for breakpoints: the execution of the instruction at a specified address, the use of data at a specified address, the use of data in a specified range of addresses (any field of a record or any element of an

array), the use of a specified data value, the use of a specified addressing mode under specified conditions (use of a specified local variable in a specified function, regardless of the value of the stack pointer when the function is called), occurrence of a specified sequence of subevents, occurrence of a specified sequence of subevents without other specified subevents intervening, etc. With appropriate hardware support (e.g., an in-circuit emulator (8)), all these and more can be breakpoint events while the program runs at full efficiency. Unfortunately, adequate hardware support is rarely available (e.g., on every processor of a parallel computer), and interpretive execution is rarely acceptable, so most debuggers have had to make do with only the first kind of event, control passing through a certain address.

The simplest breakpoint mechanism is to compile a call to the debugger, as a subroutine, at the point of interest. The debugger as a subroutine is often considered unsatisfactory because the only options after a breakpoint are to quit or to resume; however, this restriction would probably apply in any case because the ability to restart execution anywhere, familiar from assembler level debuggers, is difficult to make work for high level languages. More seriously, the debugger as a subroutine is regarded as unsatisfactory because it imposes the slow edit-compile-link-download-execute cycle in the debugging process, so breakpoints that can be planted and removed as debugger operations are demanded instead. It should be noted that the debugger as a subroutine implies that it uses stack space, and we have already observed that stack overflow is a serious problem. If breakpoints are compiled in, then the stack bouncer will allow for them as for any other procedure, but if they can be planted at run time, the worst case assumption of increasing the size of every stack is necessary, which can be costly if there are many lightweight tasks. Alleviating this problem suggests that the debugger should instead be a separate task, so a breakpoint only needs message sending stack space. This solution is attractive in that it also facilitates enhanced functionality, and so we use it with the Harmony debugger, nevertheless it does have drawbacks. For example, it means that breakpoints cannot be set in the lowest level of interrupt handling, because that is below the task abstraction and message passing can only be used within the abstraction. Using standard system services for handling breakpoints complicates the basic requirement of saving the task state so it can be examined and later restored, as the state that might change is more than just a few registers — to avoid this, some operating systems essentially duplicate their normal intertask mechanisms just for the debugger.

The usual method of planting breakpoints involves replacing the instruction to be breakpointed with a trap to a toehold of debugger code. The toehold can send a message to the debugger task, which then can determine whether the breakpoint applies to this task, can execute planted code, can initiate an interactive dialog, and so forth, but eventually will probably want to return from the breakpoint. To do this requires that the breakpointed instruction either be emulated or be replaced and executed in a controlled manner so that the breakpoint can be replanted immediately. When implementing breakpoints on a multitasking multiprocessor, there are some critical races that must be allowed for. Higher priority tasks can preempt a task that has tripped the breakpoint even while it is executing the toehold or emulating the breakpointed instruction, and of course they too may hit the breakpoint. It turns out that the most delicate critical races concern deleting planted breakpoints, and we have not found a satisfactory way to do this from another processor that shares address space. Instead we have been forced to use an agent task on the affected processor or to integrate this operation into the operating system kernel.

While the machinery for breakpoints exists to support interactive examination of the program, it can be used in a more general way to provide for planted code, of which a call to the interactive debugger is only one possibility. The immediate benefit of this approach is that it provides conditional breakpoints, where the interactive debugger is not entered every time control passes the trigger address, but only when some computable condition is satisfied. Another benefit is making temporary changes, especially repairs, to a program, thus deferring the edit-compile-link-download cycle, although this must be used with discipline so the changes are properly reflected in the source code. Planted code is often implemented as interpretive text executed within the context of the debugger, thus avoiding aggravating the stack overflow problem (and incidentally often fitting in more easily with a program compiled from a high level language). There are some interesting scoping questions about planted code, as to whether variables are evaluated in the context of the debugger or the context of the point at which the code is planted.

Where can breakpoints be placed? Users familiar with assembly level debuggers often want the ability to do single stepping of their programs. It is not always clear what this means in many high level languages, but there are examples of debuggers for such languages that do provide effective source level debugging. Unfortunately, most commercial compilers, especially those with significant optimization, leave little correspondence between the source level program and the machine language image that executes (9) — and provide inadequate symbol tables and no debugging schema to assist symbolic debugging. Function calls and returns are usually identifiable, and can be used as possible breakpoints, as sometimes can basic blocks, especially those with explicit labels. Often this is enough, which is perhaps not surprising, given that tracebacks of the stack are recognized as useful information for a debugger to display. This is also fortuitous, because in general, without help from the compiler, doing any more requires understanding and working with the compiler generated machine code. Working with compiler generated machine code is only practical with the aid of a disassembler.

The previous discussion of breakpoints focussed on what happens at a single breakpoint but, particularly when trying to understand performance issues or locate performance bugs, it is often useful instead to look between successive breakpoints or across several. A simple timer started by the resume command of the debugger and stopped when the next breakpoint occurs is a very effective tool for assessing particular critical paths. Program profiling can be done other ways, but building a profiler on a debugger, with counts and times taken at breakpoints, is attractive, especially when not all the source code is available, for instance when working with proprietary libraries.

Let us turn to the interactive aspect of the debugger. We have already indicated the importance of displaying state, and most state is recorded in complex data structures. While almost any debugger can display satisfactorily the value of a simple scalar variable, few will display an isolated record or an array in an intuitive way, and the insightful display of complex data structures is a real challenge (10). Often the values the user needs to see are not directly available but must be derived. The first step in this is for the debugger to provide expression evaluation. These expressions should not just involve literal values, but also variables from the program, and debugger variables that provide a way to use values from earlier breakpoints. This statement implies that it should be possible to save expression values in debugger variables. It is also valuable to be able to store them in program variables. Patching values this way is of tremendous significance in its primary use to steer control flow: it is often the only practical way to explore error handling code when the errors are rare. It has a secondary use when an incorrect computation has been identified — if the correct input for the next stage of the computation can be patched in, it may be possible to defer the edit-compile-link-download-execute cycle until after the next problem is uncovered.

Expression evaluation is only the first step, however, and we have already described the significance of audit functions. Traditionally these are supported by being able to call procedures from the debugger. Again there are interesting scope problems, as to whether these are evaluated in the context of the debugger or the context of the breakpointed task. Other than supporting audit functions, calling procedures from the debugger has another important application, in that often the best way for the programmer to provide a problem-specific way to display a complex data structure is to provide a procedure the debugger can call to display it.

Many assembler level debuggers provide the ability to search memory for occurrences of specific (possibly multi-word) data. This has a role in high level language debugging also. When data structures are dynamically allocated, it may be easier to search the heap directly for a record you know should exist than to chase every possible chain it might be on. A search of the heap may be essential if you know the record has just been unlinked from a chain but has probably not yet been linked somewhere else or reallocated.

5. FEATURES FOR REALTIME OR PARALLELISM

Whereas features useful for debugging sequential programs are relatively well understood, features for debugging realtime or parallel programs are still very much a research area, and there has been little experience with such tools (11). Here are the things we have found useful, either ad hoc or as part of the Harmony debugger.

In preceding sections we discussed how the decreased role of control flow and the increased significance of understanding state resulted in the importance for multitask programs of examining standard system data structures, and we also discussed audit routines as preferable to manual examination. Exactly because these data structures are important and are standard, and because their noteworthy aspects are also standard, audit and display routines for them should be provided with the debugger.

The first significantly different tool of realtime or parallel debugging, invented seemingly independently by each group of programmers with extensive realtime or parallel debugging experience, is the history buffer. To avoid having to rerun the program to learn how some situation occurred, with consequent questions of determinism or feasibility, we simply record significant events as they happen, so that we can go back and look at them later, if necessary. Normally this log is kept in a circular buffer, overwriting events no longer of interest, thus avoiding unbounded storage requirements and possibly even the costs of external I/O. Sometimes multiple buffers are kept with different wrap rates. What is written to a history buffer is a sequence of records, giving event type and parameters. The question of what events to write to history buffers is clearly up to the programmer and should be available both compiled in and planted, just like breakpoints. A history buffer can be read from the debugger without stopping the logging tasks. Experience indicates that recording too much is even worse than recording too little, as the log may not be long enough and even if it is, filtering out the events of interest can be too difficult. It is worth noting that some people have explored correlating events from disjoint history buffers to reduce interaction artifacts from too many tasks contending over the same shared history buffer, and also that interactive dynamic graphics has been found useful to replay events leading up to something.

A tool similar to the history buffer, but which trades resolution in time for reduced synchronization overhead, is the vector of use bits. The name comes from the use bits and dirty bits that often support demand paging systems, providing an inexpensive record of access. A use bit in our context is explicitly cleared and is set by a compiled in or planted command. Often it is useful information to know that some piece of code, containing the set command, was not executed since the bit was cleared — and equally it is often useful information to know that the piece of code was executed at least once. A bit in the use bit vector might, for instance, be set in the code for a function so we can find if that function was ever called, or another bit might be set in the code for a task so that we can monitor to find how frequently the task is dispatched. Sometimes clearing and reading a group of use bits must be atomic to be useful.

It would be nice to have something like use bits for data structures, and of course we could set a use bit everywhere the data structure is referenced, but for many data structures this would be impractical, even if we just tried to identify where the data structure was changed. A tool that can be valuable is the ability to checksum an arbitrary block of memory. Checksumming blocks of memory (possibly containing many records) is useful for confirming that nothing has changed since the last time the block was checksummed, or for checking that something has actually changed as a precursor to a detailed exploration of the changes. There are some difficulties, however, in deciding what it means to say a data structure was unchanged. Is a list structure unchanged if the heap from which the records come is compacted, with pointers updated appropriately? If a record is deleted from a singly linked list by copying its successor over it, is the successor unchanged? Demanding this kind of equivalence makes monitoring difficult, even with hardware assistance.

Under features for debugging sequential programs we discussed the advantages of being able to call procedures from the debugger. The major problem in a multitask situation is stack overflow again: as with planted breakpoints, the possibility of procedures called from the debugger cannot be predicted by the stack

bounder. The solution is again the same, i.e., in a multitask program, instead of calling procedures from the debugger, we need to create and destroy tasks from the debugger and most importantly we need to be able to create request messages, send them to such tasks, and interpret the reply messages. This facility is also useful for probing actors, such as the administrator task mentioned in the section on troubleshooting. It can also be used for stimulating execution of exception handling since in a multitask program exception handling is often done by separate tasks.

We have already indicated that performance problems are a major concern in realtime and parallel computing. Load balance can be part of this. One way to detect problems, especially when tasks do not migrate between processors, is to look for excessive idle cycles on a processor. In many multitask systems an idle task runs when a processor has no other work — if the idle task increments a counter that can be monitored by, say, a spy task, then excessive idle cycles can be detected easily.

For many message passing systems, the suggestion has been made that a useful debugging technique would be to intercept the messages, or to tap the I/O streams. This would be technically difficult to do in Harmony for two reasons. First, the use of remote procedure call semantics instead of asynchronous message sending means that the sender or receiver may, and often does, depend on the blocking behaviour of the other, and another task inserted between them might have difficulty duplicating that behaviour. Second, the fact that messages in Harmony are sent directly to a specified task_id instead of indirectly through some named channel, say, means there is no central place where intercepting the selected messages would be easy. We have not regarded this as a problem because our history buffers provide similar facilities.

6. USER INTERFACE ISSUES

All too often debuggers are unused because they are hard to learn and remember or because their output is confusing. Just distinguishing the dialogue you are having with the debugger from the dialogue you are having with the program being debugged is often a trial. Ever thought about using a traditional glass teletype oriented debugger to debug a new version of the debugger itself while it is being applied to some program? The user interface can make or break a debugger.

Program development requires learning a mass of details about the language, the libraries, the text editor, the problem specification, and so forth, and these details are memorized by continually working with them. If a debugger is to be powerful, that is, if it can provide needed facilities succinctly instead of forcing the user laboriously to concoct them manually from primitive operations, then the debugger interface must be rich, which, if it is command driven, means the user has a lot of notation and syntax to learn. Debuggers are not used as frequently as some other program development tools, however, so few programmers learn them well. The design of some debuggers attempts to reduce the burden by using the syntax and semantics of the language being debugged where possible in commands to the debugger, and similar benefits can be gained by the debugger being cognizant of system semantics. There are still gaps, however, in that one wants to do things with debuggers that are outside the language (such as to examine variables not currently in scope), and stretching semantics sometimes produces nonintuitive idioms or even clashes. Striking benefits have been seen when the debugger interface is driven instead by the modern facility of context sensitive pop-up menus (12, 13), where the user chooses only among options possible at the current point. People have even productively used such a debugger without being aware that a manual exists!

Even more important is the effect of the modern facility of multiple windows. Multiple windows are always nice, but one might go so far as to assert that for satisfactory debugging of multitask programs they are essential. A number of features matter. First is the management of screen real estate. There is a lot of information a debugger can extract from a program, much of it only intelligible with highly structured display. The screen is never large enough, and if it were, things of interest might be too widely separated by clutter. It is not just that a scrolling glass teletype dialogue, losing items off the top, is unsatisfactory, but also that fixed form screens are inadequate. The user at the screen must have the ability to rearrange displays to bring together items he wants to focus on. He must also have the power of elision, to hide displays not currently of interest,

yet be able to show those displays again when his focus changes. A second feature of importance is that the user should never have to retype data already displayed somewhere on the screen. Over the course of a debugging session so much has to be specified, detail is so important, and typing errors are so easy, that in traditional debuggers repeated attempts to get particular information are common, which is especially frustrating when that information had been displayed before! This problem is largely finessed by a pointing-based copy-and-paste ability. A third feature, alluded to in the opening of this section, is that separate windows for debugger dialogue and application dialogue immediately solves a traditional headache. This can be carried further by using disjoint windows for other functions such as symbol tables, cross references, and source code display. It is the fourth feature, however, that clinches the issue, and that is that separate windows can be used for the dialogues associated with debugging separate tasks. Because we recommend that debugger directives be associated with specific tasks and because tasks proceed asynchronously, a single debugger dialogue quickly loses track of less frequent activities.

Of course, ideally, the debugger dialogue should involve communication with the user at the level at which he thinks about his program, which is often described as symbolic debugging. Identifying variables and procedures by their symbolic names, relying on declared type information to automatically generate data structure display, and setting breakpoints or planting code in terms of the original source text contribute to this. We must note, however, that this ideal is not always achievable. There can be problems with the application itself. Deficiencies in the programming language may mean that the programmer actually thinks about his program in abstractions above the level of the language (dynamic storage management or use of records in Fortran, for instance), so it is desirable if at least the debugger lets such constructs exist. On the other hand, detailed pursuit of something (a timing dependent bug or a compiler bug, for instance) may force the programmer to think about his program at a much lower level than the language in which he wrote it, and the debugger needs to support that too.

More seriously, the ideal of communication at the user's level may not be achievable because of the environment. We already observed that most compilers do not support this because the symbol tables they produce are inadequate and they produce no debugging schema. If we want to be portable across development systems, this problem cannot be dismissed by sticking to a compiler that is good enough. A completely separate tool to process the source text and with access at most to load maps can alleviate but not remedy the problem. Even if we had no compiler problems, however, we still might not achieve the ideal. Most realtime computing, especially for embedded systems, is done on a target machine that is different than the host on which development was done, so the source code and other debugging information may not be available at run time and the original development host may not be attached. Proprietary libraries that are available only in object form have much the same effect. Production code distributed to the field, whether in embedded systems or mass market commercial products, again often exhibits this effect. In short, we cannot always count on source code being available nor on the executable image including debugger support, so we are forced to think of the debugger itself having access to a file system and building up there the information we need. Relying on a microcomputer as a smart terminal for the debugger is one approach.

The preceding discussion has brought out a point that is often overlooked, that serious debugging requires persistent context and conversely that persistent context can facilitate debugging. The simple observation is that a single snapshot of a program is not enough and that breakpoints tripped on instantaneous events only are not enough. We have already observed that debugger variables, to record values found earlier, qualitatively enhance expression evaluation and value patching, and that sequential triggering of breakpoints qualitatively enhances the ability to locate protocol bugs. Now we observe that understanding a complex program, debugging it and tuning it, does not happen in a single debugging session and that retaining context from one debugging session to the next can save a tremendous amount of time. Whether or not the executing program is saved and restarted, there is enormous value in the debugger being able to save things it has learned and read them to restart the next time.

The final user interface topic we want to touch on is how the debugger gets associated with the program in the first place. Traditionally, debuggers were linked in with test versions of programs, or the debugger was

called as a command that then invoked the program it was to control. What this means is that when the program running in production fails, perhaps because of a rare and unanticipated event, the debugger is not available as a tool to locate the problem and get around it. When the debugger is a separate task, the cost of having it present even when the program is running in production is often small compared to the potential benefit. A programmer might even occasionally dial up such a program by remote access to monitor the program's health. Of course, in general, there would be no programmer conversing with the debugger as the program runs, and even if a human operator was interacting with the program when it suddenly took an exception, there is no guarantee that he would know what to do, or that adequate support information to recover would be available. Nevertheless, automatic entry into the debugger is a better last response than the traditional abnormal termination message!

7. CONCLUSIONS

When working in the right programming abstraction, supported by the right tools and techniques, debugging multitask programs is no worse than debugging any other programs.

REFERENCES

1. G. Sande. Personal communication. 1971.
2. R.L. Glass. Persistent software errors. *IEEE Trans. on Software Eng.* 7(1): 162-168; 1981.
3. P. Abrahams. The role of failure in software design. *Commun. ACM*, 29(12): 1129-1130; 1986.
4. R.L. Glass. Patching is alive and, lamentably, thriving in the real-time world. *ACM SIGPLAN Notices*, 13(3): 25-28; 1978.
5. W.M. Gentleman. Using the Harmony operating system. NRC/ERB-966, National Research Council of Canada, Ottawa, Ontario, December 1983, revised March 1987.
6. W.M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software Pract. Exper.* 11(5): 435-466; 1981.
7. D.R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. North Holland, New York, 1982.
8. W.M. Gentleman and H. Hoeksma. Hardware assisted high-level debugging. *J. Syst. Software*, 3(4): 309-314; 1983.
9. P.T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. Ph.D. Dissertation, Computer Science Division—EESC, University of California, Berkeley, 1984; also Report CSL-84-5, Xerox PARC, Palo Alto, CA, May 1984.
10. B.A. Myers. Displaying data structures for interactive debugging. Report CSL-80-7, Xerox PARC, Palo Alto, CA, June 1980, reprinted June 1982.
11. M.S. Johnson, ed. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*. Pacific Grove, CA, March 20-23 1983; *SIGPLAN Notices*, 18(8): 1983.
12. T.A. Cargill. The Blit debugger. *J. Syst. Software*, 3(4): 277-284; 1983.
13. T.A. Cargill. Implementation of the Blit debugger. *Software Pract. Exper.* 15(2): 153-168; 1985.